# Selecting Representative Benchmark Inputs for Exploring Microprocessor Design Spaces

MAXIMILIEN B. BREUGHE and LIEVEN EECKHOUT, Ghent University, Belgium

The design process of a microprocessor requires representative workloads to steer the search process toward an optimum design point for the target application domain. However, considering a broad set of workloads to cover the large space of potential workloads is infeasible given how time-consuming design space exploration typically is. Hence, it is crucial to select a small yet representative set of workloads, which leads to a shorter design cycle while yielding a (near) optimal design.

Prior work has mostly looked into selecting representative benchmarks; however, limited attention was given to the selection of benchmark *inputs* and how this affects workload representativeness during design space exploration. Using a set of 1,000 inputs for a number of embedded benchmarks and a design space with around 1,700 design points, we find that selecting a single or three random input(s) per benchmark potentially (in a worst-case scenario) leads to a suboptimal design that is 56% and 33% off, on average, relative to the optimal design in our design space in terms of Energy-Delay Product (EDP). We then propose and evaluate a number of methods for selecting representative inputs and show that we can find the optimum design point with as few as three inputs.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*Modeling of computer architecture*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling techniques*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Processor design space exploration, workload selection, input selection

**ACM Reference Format:**
Breughe, M. B. and Eeckhout, L. 2013. Selecting representative benchmark inputs for exploring microprocessor design spaces. ACM Trans. Architec. Code Optim. 10, 4, Article 37 (December 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555294



## 1. INTRODUCTION

Designing a microprocessor core involves making a large number of tradeoffs. Computer architects need to decide on a large number of design choices such as core type (out of order versus in order), pipeline depth and width, clock frequency, cache size and organization, branch predictor type and configuration, various buffer sizes, and so forth. Making these design choices not only involves considering performance but also requires taking into account other criteria or constraints such as energy and power consumption, temperature, chip area, cost, reliability, and so forth. In fact, for many of today's systems, these constraints are at least as equally important as performance, if not first-order design targets. The huge design space along with the complex, multi-objective optimization criteria makes design space exploration a challenging problem.

This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259295.
Authors' addresses: M. B. Breughe and L. Eeckhout, ELIS Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: mbreughe@gmail.com, lieven.eeckhout@elis.ugent.be.

DOI: http://dx.doi.org/10.1145/2555289.2555294

To make things even worse, architects and designers typically rely on detailed cycle-accurate simulation during the exploration. In other words, the evaluation of each possible design point under consideration involves time-consuming detailed simulation, which makes the whole design process extremely challenging, especially given the short time to market.

Exploring the design space reliably requires representative workloads. Workloads that are nonrepresentative for the envisioned target workloads may lead to suboptimal designs, for example, designs with poor performance and/or excess power consumption. In other words, the final design hinges on the workloads used during the exploration and design cycle. The importance of having representative workloads is well known, and several prior works have proposed methods to evaluate benchmark representativeness; see, for example, Eeckhout et al. [2002] and Yi et al. [2003]. Next to being representative, a benchmark suite should also be relatively small: Simulating a very broad set of benchmarks is not only time-consuming but also requires a lot of simulation resources, which can be costly. Hence, the key challenge when composing a benchmark suite is that it should be relatively small while being representative.

Most of the prior work on the topic of workload selection has focused on selecting representative benchmarks, and limited effort has been geared toward exploring the impact of the benchmark *inputs*. While runtime behavior may be relatively insensitive to the inputs for some applications, other applications may be very sensitive. For example, a streaming application, such as a filter operation, may be largely input insensitive as the application executes the same code and accesses memory in a highly predictable way no matter what input it is given. On the other hand, a video application that decodes an action movie with lots of complex scenery versus a recording of a news reader with little variation across subsequent frames is likely to lead to different code regions being executed, as well as different memory access patterns being observed, which in turn leads to different runtime behavior.

In this article, we quantify how sensitive design space exploration is to benchmark inputs, and we subsequently present methods for identifying representative inputs for reliable design space exploration. Using 1,000 inputs for a set of 20 embedded benchmarks while considering a core microarchitecture design space with approximately 1,700 design points, we find that selecting a single randomly chosen input per benchmark may (in a worst-case scenario) lead to a design point with an Energy-Delay Product (EDP) that is 56% worse, on average, compared to the optimum design point (the design point with the minimum EDP in our design space for the benchmarks and inputs considered in this study); three randomly chosen inputs may still lead to a design point that is 33% worse. We then present three methods for selecting representative benchmark inputs, with each method representing a different tradeoff in time complexity versus accuracy. The most accurate method identifies three representative inputs that lead to the optimum design point during design space exploration. The fastest method, which requires a single, relatively fast profiling run for each potential input to identify two representative inputs, yields a design point that is within 3.7% on average compared to the optimum design point.

Overall, we make the following contributions in this article:

—We study how sensitive microarchitecture design space exploration is with respect to benchmark inputs. Whereas prior work focused on selecting representative benchmarks, this article is the first to comprehensively explore the impact of benchmark inputs during design space exploration. We find benchmark inputs to have a significant impact on the final design point.
—We quantify that the common practice of randomly selecting a single (or few) benchmark input(s) may lead to suboptimal designs with significant deficiencies compared to the optimal design point.

Table I. Design Space for Detailed Simulation

| | conf0 | conf1 | conf2 | conf3 | conf4 |
|---|---|---|---|---|---|
| Issue width | 2 | 4 | 4 | 2 | 4 |
| Data cache size | 16KB | 64KB | 8KB | 64KB | 64KB |
| Instruction cache size | 16KB | 8KB | 32KB | 16KB | 32KB |
| Branch predictor | 1KB global | 1KB global | 3.5KB hybrid | 1KB global | 3.5KB hybrid |
| Pipeline depth | 9 | 9 | 9 | 7 | 9 |

—We present and evaluate several methods for selecting representative inputs, making the design space exploration more reliable and leading to better performing design points. These methods vary in the way they characterize inputs, leading to varying degrees of time complexity, as well as representativeness of the selected inputs.

We believe this work is both important and timely. Given the end of Dennard scaling [Dennard et al. 1974] in the presence of continued transistor integration via Moore's Law, there is a trend toward specialization in order to keep improving performance and energy efficiency [Esmaeilzadeh et al. 2011; Hameed et al. 2010; Venkatesh et al. 2010]. Designing application-specific processors and/or accelerators is a promising and fruitful avenue in this direction. This work quantifies the importance of representative benchmark inputs to the quality of the final design point and provides a method for identifying representative inputs when specializing a processor design for a particular application. We hope this work will help researchers and developers design high-performance yet energy-efficient processors across a broad range of workloads by paying attention to having representative benchmark inputs.

## 2. MOTIVATION

We first set up a limited experiment to further motivate the problem and gain some initial insight before diving into a systematic evaluation of the importance of benchmark inputs during design space exploration. We consider one benchmark from the MiBench benchmark suite [Guthaus et al. 2001], namely, sha, a secure hash algorithm, with five inputs and five processor configurations. The five inputs were randomly selected from the 1,000 input sets provided through KDataSets [Chen et al. 2010], and the five processor configurations are shown in Table I; we consider superscalar in-order processors and vary issue width, data and instruction cache size, branch predictor configuration, and pipeline depth. We simulate these processor configurations using the `gem5` simulator [Binkert et al. 2011].

Figure 1 reports normalized EDP for each of the five inputs across these five processor configurations, along with the average EDP across all inputs. The EDP values are normalized against the processor that is optimal for the given input. Hence, an EDP value of 1 denotes the optimal processor configuration for a given input (e.g., configuration 0 for input C), and the closer the normalized EDP values are to 1, the better. It is immediately clear from the results shown in Figure 1 that configuration 0 is the most optimal processor configuration (i.e., the one with the lowest EDP) across all five inputs: For most inputs, its EDP is close to 1 and it has the lowest EDP on average (see the "average" bars in the graph). The same result would be obtained in case inputs A, B, or C would have been chosen to guide the design space exploration. However, using inputs D or E during design space exploration would have led us to believe that configuration 4 is most optimal. The risk now is that if configuration 4 were to be deployed in the field, an end-user using another input, for example, input C, would experience a 73% worse EDP compared to configuration 0. The pitfall here is that it is unclear at design time whether we are in the case of inputs D or E or one of the other inputs. In other words, using a nonrepresentative input to drive the design

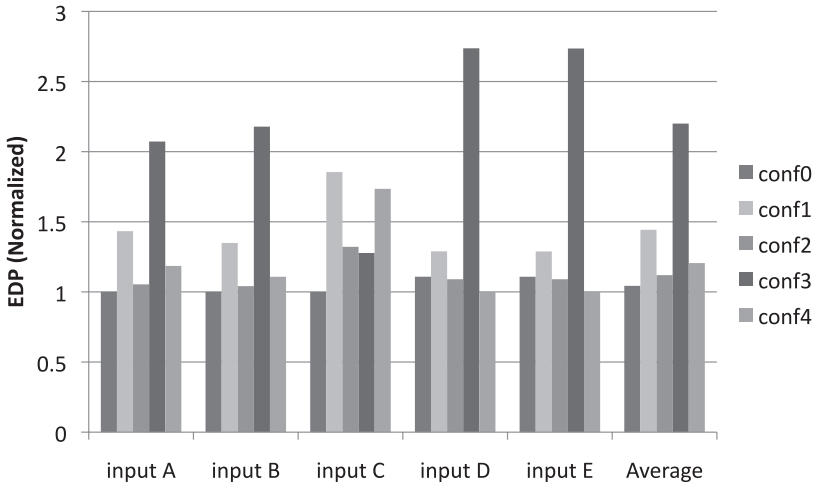


Fig. 1. Normalized EDP for sha for five different processor configurations and five different inputs.

Table II. Design Space Considered in This Study

| *Parameter* | *Range* |
|---|---|
| Issue width | 1 - 2 - 3 - 4 |
| Data cache size | 8KB - 16KB - 64KB |
| Data cache associativity | 2 - 4 |
| Instruction cache size | 8KB - 16KB - 32KB |
| Instruction cache associativity | 2 - 4 |
| Cache block size | 32 - 64 byte blocks |
| Branch predictor | 1KB global - 3.5KB hybrid |
| Pipeline depth | 5 - 7 - 9 |

space exploration may lead to a design point with poor performance for other inputs, and this is hard to know a priori (if at all possible) without doing a full exploration.

This case study motivates the need for a methodology for identifying representative inputs for design space exploration. The remainder of the article will further investigate and quantify the pitfall of nonrepresentative inputs in a systematic way and will present methods for identifying representative inputs so as to minimize the chance of ending up with a design point during design space exploration that could lead to suboptimal and even poor performance for unseen inputs when deployed in the field.

## 3. EXPERIMENTAL SETUP

We now give a detailed overview of the design space and the workloads considered in this article. We also describe our modeling infrastructure to efficiently explore the huge design space, as well as the optimization criterion we are targeting.

### 3.1. Design Space

The design space considered in our exploration is shown in Table II. We assume a superscalar in-order processor core and we vary instruction and data cache size, associativity and line size, pipeline depth, issue width, and the branch predictor configuration. We consider two to four options along each of these axes. The cross-product of all these design options leads to a design space consisting of 1,728 microarchitectures. This is a fairly small design space compared to real-life design spaces, yet it enables us to illustrate our contribution while being manageable in terms of time complexity. The

Table III. Overview of Benchmarks

| *Benchmark* | *Suite* | *Category* | *Description* |
|---|---|---|---|
| adpcm_c | MiBench | Telecommunication | Pulse code modulation with ADPCM (encode) |
| adpcm_d | MiBench | Telecommunication | Pulse code modulation with ADPCM (decode) |
| dijkstra | MiBench | Network | Shortest path calculation between nodes in a graph |
| gsm | MiBench | Telecommunication | Voice encoding with GSM |
| jpeg_c | MiBench | Consumer | Lossy image compression with JPEG standard |
| jpeg_d | MiBench | Consumer | Decompression of JPEG compressed images |
| lame | MiBench | Consumer | MP3 encoding |
| patricia | MiBench | Network | Compression of network data structures |
| qsort | MiBench | Automobile/Industrial | Data sorting (e.g., 3D coordinates) |
| rsynth | MiBench | Office | Text-to-speech synthesis |
| sha | MiBench | Security | Secure hashing for, e.g., digital signatures |
| stringsearch | MiBench | Office | Searching of words in phrases (case insensitive) |
| susan_c | MiBench | Automobile/Industrial | Corner recognition in images |
| susan_e | MiBench | Automobile/Industrial | Edge recognition in images |
| susan_s | MiBench | Automobile/Industrial | Smoothening of an image |
| tiff2bw | MiBench | Consumer | Conversion of a TIFF image to black and white |
| tiff2rgba | MiBench | Consumer | Conversion of a TIFF image into RGB-formatted TIFF |
| tiffdither | MiBench | Consumer | Dithering of a black-and-white TIFF image |
| tiffmedian | MiBench | Consumer | Conversion of an image by reducing its color palette |
| h264 | CPU2006 | Video Compression | Video compression according to H.264/AVC |

reason for choosing in-order processors is that in-order processors are inherently more energy efficient compared to out-of-order processors, which is in line with the workloads selected (as described next) and our goal of exploring application-specific processors.

### 3.2. Workloads

Table III provides an overview of the benchmarks used in this article. All but one of the benchmarks are taken from MiBench [Guthaus et al. 2001]. We use the MiBench benchmark suite for two reasons. First, MiBench is targeted toward embedded processors and devices, which aligns well with the goal of tuning processor architectures for a specific workload domain. Second, there exists a large set of inputs for each of these benchmarks, which we use to explore input sensitivity. The inputs were taken from the KDataSets database [Chen et al. 2010], which provides 1,000 inputs per benchmark. These inputs have a working set size that is too large to fit in the processor core's caches; that is, the data working set is typically larger than the 64KB data cache considered during the exploration. Using 1,000 inputs per benchmark leads to significant simulation time requirements for the experiments in this article; nevertheless, a sufficiently large number of inputs is needed to quantify the impact of input selection on design space exploration. Chen et al. [2010] found these inputs to be diverse based on a detailed characterization using both microarchitecture-dependent and -independent metrics.

In addition to these MiBench benchmarks, we also include the h264 benchmark from SPEC CPU2006 [Henning 2006] as it is highly relevant for our application domain of interest while exhibiting interesting dynamic, time-varying behavior not observed in the MiBench benchmarks. Because CPU2006's h264 is not part of the KDataSets inputs database, we had to create our own set of 1,000 video inputs. We took 50 raw video files from the public domain [Xiph.org 2012] and generated 4 to 11 different sequences for each of these videos by selecting different begin and end points, leading to 250 different video sequences in total. For each video sequence, we generated four random encoding schemes by varying 48 different parameters. Parameters include varying the number

of $B$ frames; quantization parameters for $I$, $P$, and $B$ slices; enabling/disabling pyramid encoding; and so forth. The end result is 1,000 video inputs for the h264 benchmark.

We compiled all of our benchmarks using the gcc-4.3 cross-compiler for the Alpha ISA. Our default optimization flag is -O3. We also evaluate the impact of compiler optimization flags on processor architecture design space exploration in one later section in the article. We therefore consider 250 randomly chosen combinations of compiler optimization flags. Random selection of compiler optimization flags was previously shown to give a fairly good coverage of the impact of compiler flags on overall performance, and 250 combinations was found to be sufficient to achieve near-optimal performance [Chen et al. 2010].

### 3.3. Modeling Infrastructure

Because of the very large number of measurements needed in this study (the processor design space contains 1,728 design points, and there are 20 benchmarks and 1,000 inputs per benchmark—a total of more than 34 million measurements), using detailed cycle-accurate simulation (e.g., using gem5 as done in the motivation section) would have been totally infeasible.

Instead, we resort to a previously proposed mechanistic analytical model for estimating performance [Breughe et al. 2012] in order to collect this huge number of measurements in a reasonable amount of time. The model targets superscalar in-order processors and models the performance impact of issue width, pipeline depth, nonunit instruction execution latencies, interinstruction dependencies, cache/TLB misses, and branch mispredictions. It predicts performance within 2.5%, on average, compared to detailed cycle-accurate simulation. The model takes as input a number of program statistics to characterize an application's instruction mix, interinstruction dependencies, and cache and branch behavior. This collection is a one-time cost per benchmark and input. Although we need to characterize each benchmark/input pair, the characterization is much faster than detailed simulation. The other model inputs relate to the processor architecture being considered, such as pipeline depth, width, instruction latencies, and so forth.

We use McPAT to estimate power consumption [Li et al. 2009]. The inputs to McPAT are various processor configuration parameters, such as pipeline depth, width, cache configuration, memory latency, chip technology (32nm), and so forth, along with program parameters, such as the number of dynamically executed instructions, the instruction mix, and so forth, and finally, program-machine parameters, such as number of cache misses, branch mispredictions, and so forth.

Put together, performing all experiments reported in this article (including program profiling, running the performance model, and collecting power estimates) required 12,000 compute days on a single machine, or, in our case, 40 days on a cluster with 300 machines.

### 3.4. Optimization Criterion

Although the problem we want to tackle is fairly easy to state, attacking it in a systematic way is rather complex. There are multiple dimensions involved in our study: we consider multiple benchmarks, and multiple inputs per benchmark, each leading to a different dynamic instruction count; we consider multiple microarchitectures; and we consider two optimization criteria, namely, performance and power/energy.

In order to accurately and confidently evaluate how well a limited set of (presumably) representative inputs, selected by our methods, captures the behavior of the complete input database, we need to have an appropriate metric. We quantify a microarchitecture's energy efficiency using the EDP, which is computed as the total energy consumed multiplied by the total execution time to perform a given unit of work (i.e., the

complete execution of the benchmark with its input). Because we have multiple inputs per benchmark, we need an appropriate way of computing the average EDP across these inputs to obtain the EDP for a particular benchmark and microarchitecture. We also need a way of comparing the EDP of the presumably optimal processor determined using a limited set of inputs against the EDP of the optimal processor configurations determined using all inputs. This is done as follows:

(1) Calculate the execution time $T(i, j)$ for each input $i$ and microarchitecture $j$. This is done using the mechanistic analytical performance model previously described.
(2) Calculate the consumed energy $E(i, j)$ for each input $i$ and microarchitecture $j$. This is done using McPAT.
(3) Compute the execution time and energy consumption per instruction. This is done by dividing execution time and energy consumption by the number of dynamically executed instructions $I(i)$, following Equations (1) and (2):

$$TPI(i, j) = \frac{T(i, j)}{I(i)}, \tag{1}$$

$$EPI(i, j) = \frac{E(i, j)}{I(i)}. \tag{2}$$

(4) Identify the microarchitecture with the minimum EDP value across all inputs. $EDP_{min}$ is computed following Equation (3), with $N$ the number of inputs in the input database ($N = 1{,}000$ in our setup):

$$EDP_{min} = \min_j \left( \sum_i^N TPI(i, j) \times \sum_i^N EPI(i, j) \right). \tag{3}$$

Note that this formula complies with the recommendations by Sazeides et al. [2005] regarding how to compute average EDP across benchmarks.

(5) We can now compute the normalized EDP ($\widetilde{EDP}$) of processor configuration $j$ relative to the minimum EDP:

$$\widetilde{EDP}(j) = \frac{\sum_i^N TPI(i, j) \times \sum_i^N EPI(i, j)}{EDP_{min}}. \tag{4}$$

The metric $\widetilde{EDP}$ thus quantifies the energy efficiency of a processor configuration relative to the optimal configuration with the lowest average EDP across all inputs and design points considered in our setup. Hence, our goal is to minimize $\widetilde{EDP}$ using as few inputs as possible during the exploration. For the remainder of this article we will refer to the design with minimum $\widetilde{EDP}$ as the optimal design. We will refer to $\widetilde{EDP}$ as "normalized EDP." Although we use this normalized EDP as our metric throughout the article, the proposed input selection methods are not constructed in a way that they are bound to using EDP as an evaluation metric.

## 4. QUANTIFYING THE IMPLICATIONS OF NONSYSTEMATIC SELECTION OF BENCHMARK INPUTS AND COMPILER OPTIMIZATIONS

Before presenting and evaluating methods for selecting a limited yet representative set of benchmark inputs, we first quantify the implications of not having a systematic way of identifying representative inputs. We do this using the experimental setup described earlier, that is, using 20 benchmark applications, 1,000 inputs per application, and while considering the approximately 1,700 microarchitecture configurations. (The prior motivation section considered just a case study with a few benchmarks, five inputs

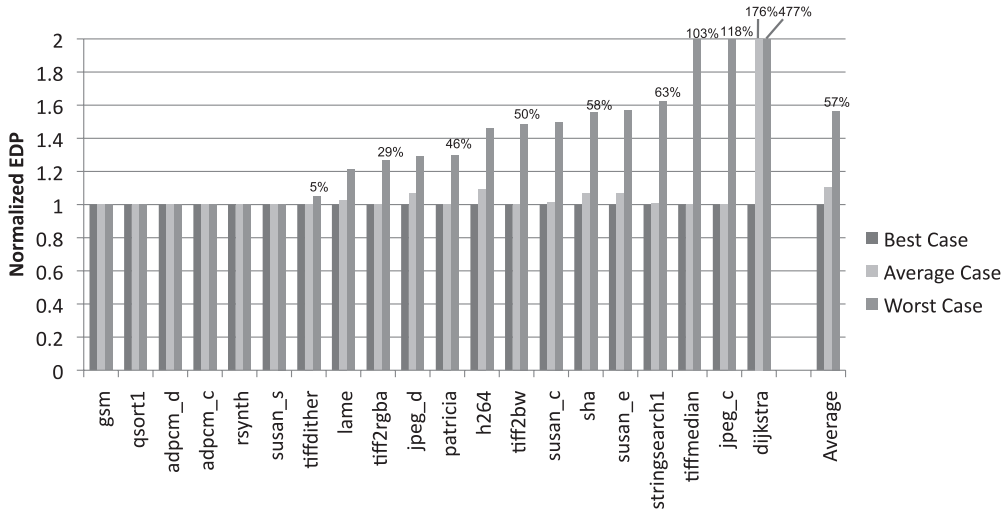


Fig. 2. Quantifying the impact of selected benchmark inputs for identifying the optimum processor configuration.

per benchmark, and five processor configurations.) In addition, we also evaluate the impact of compiler optimizations on design space exploration, and how it compares to benchmark inputs, in order to understand their relative significance, which justifies our focus on studying sensitivity to benchmark inputs.

### 4.1. Sensitivity to Benchmark Inputs

We first quantify the sensitivity of benchmark inputs on design space exploration. Figure 2 reports normalized EDP values (as defined in Equation (4)) across all benchmarks in our study while considering three scenarios. The first scenario ("best case") is an ideal scenario in which we would always pick a single benchmark input that, when used to determine the (presumably) most optimal processor configuration, would yield a configuration with a normalized EDP that is as close as possible to 1 (the normalized EDP of the globally optimal configuration across all inputs). The second scenario ("average case") is the average scenario in which we were to pick a random benchmark input for design space exploration and reports the average EDP across these randomly selected inputs. This scenario represents what is expected to happen on average. The third scenario ("worst case") is the scenario in which we would be unfortunate to pick a benchmark input that would result in a (presumably) optimal processor configuration with the worst possible EDP compared to the global optimum.

There are several interesting observations to be made here. The normalized EDP values obtained through the best-case scenario are equal to one for all benchmarks. This indicates that there exists at least one benchmark input that, when selected and used to drive the design space exploration, yields an optimal processor configuration that is indeed optimal across all inputs. The question now is whether it is possible to find, and if so, how to find such an input. As mentioned earlier, this is an ideal case, and it is hard (if not impossible) to know a priori whether a given input is going to lead to the optimal design point during design space exploration.

Picking a random input leads to a design point that is fairly competitive to the global optimum on average. For all but one of the benchmarks, we observe that a random input leads to a design point with an EDP that is close to the global optimum. For one benchmark, however, namely, dijkstra, a random input leads to a processor

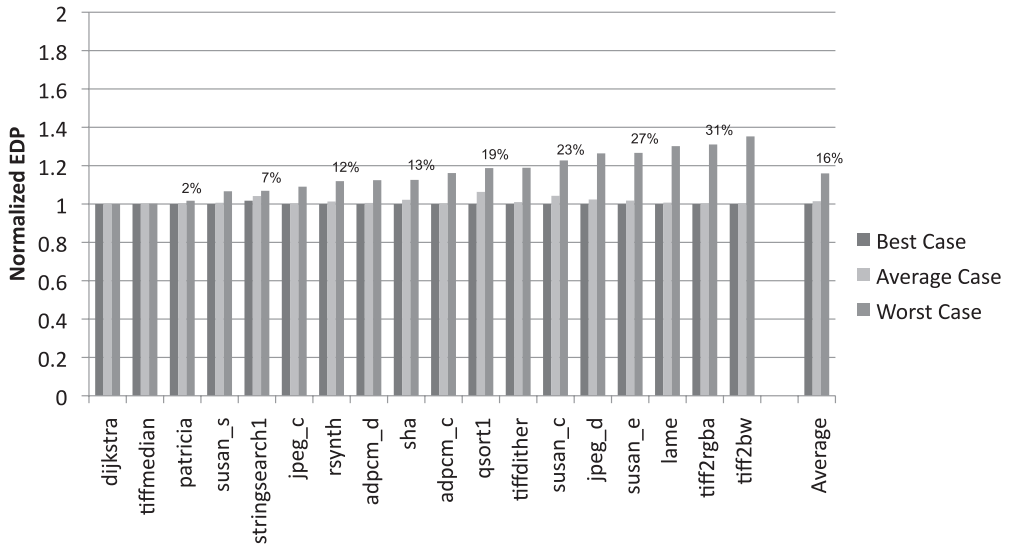


Fig. 3. Quantifying the impact of compiler optimization flags on the identification of the optimum processor configuration.

configuration with an EDP that is 176% worse on average compared to the global optimum. In other words, for some applications, selecting a single random input may lead to severe average deficiencies during design space exploration.

Picking the worst possible input leads to an EDP deficiency of 56% on average across all benchmarks, and ranges up to 477%. We observe substantial deficiencies for around two-thirds of the benchmarks; one-third of the benchmarks seem unaffected. Note that these EDP deficiency numbers are average numbers across all inputs. Whereas the average EDP deficiency for a particular benchmark across all inputs can be as high as 477%, for some inputs the deficiency can be even higher, up to 82× the EDP compared to the optimum design point. This illustrates the high sensitivity of design space exploration with respect to benchmark inputs. While the worst-case scenario is not the common case, or what is to be expected on average, it might happen, and the pitfall is that it is unclear a priori whether we have picked an input that represents the worst, average, or best case. In other words, there is no way for a designer to verify this unless one were to explore the entire design space with all possible inputs, which is infeasible in practice. We focus on the worst-case scenario in this article, for this reason. We want to avoid picking benchmark inputs that lead to suboptimal designs, and these results clearly illustrate the need for a systematic method for selecting representative benchmark inputs.

### 4.2. Sensitivity to Compiler Optimization Flags

Before presenting our methods for selecting representative inputs, we also evaluate how sensitive design space exploration is with respect to another characteristic that may affect workload behavior, namely, the compiler and its optimizations. Further, we want to understand the relative importance of input versus compiler sensitivity toward workload representativeness.

We therefore consider 250 randomly selected combinations of compiler optimization flags. (As mentioned before, random selection of 250 combinations of compiler optimization flags is found to be a robust way of measuring the performance impact of compiler optimizations [Chen et al. 2010].) According to the results shown in Figure 3,

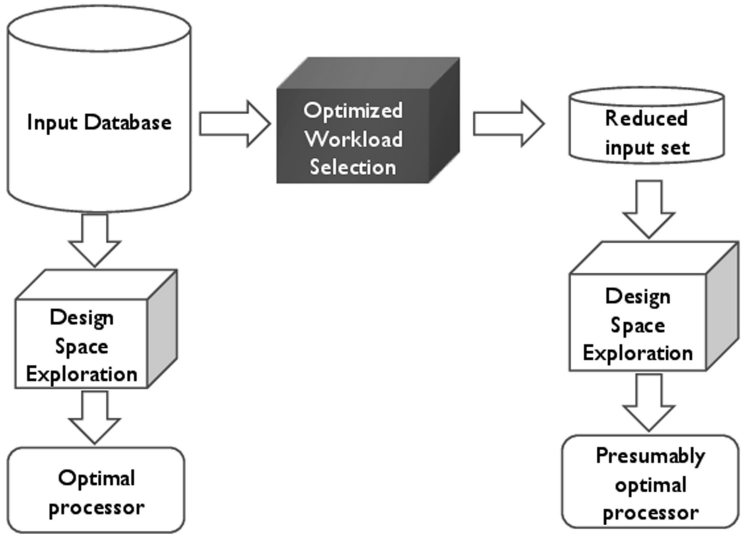


Fig. 4. Input selection workflow.

for none of the benchmarks do we observe a severe EDP deficiency in the average (and best) case scenario. We observe some EDP deficiency for some of the benchmarks in the worst-case scenario: 16%, on average, and up to 35%. In other words, the compiler optimizations used to compile the benchmarks may lead to some EDP deficiency if one were to be unlucky to select a compiler optimization that is nonrepresentative compared to the other optimizations for driving microarchitectural design space exploration. Comparing Figure 3 against Figure 2, we conclude that microarchitectural design space exploration is much more sensitive to benchmark inputs than to compiler optimizations. For this reason, we focus on the selection of representative benchmark inputs and leave the selection of representative combinations of compiler optimization flags for future work.

## 5. REPRESENTATIVE BENCHMARK INPUT SELECTION

We now present and evaluate various methods for selecting representative benchmark inputs. We do this using a common workflow as shown in Figure 4. We start off with the input database shown at the top left (i.e., 1,000 inputs per benchmark), from which we select a number of presumably representative inputs, called the *reduced input set*, shown at the top right. We evaluate all possible processor configurations for the selected inputs and determine the most optimal configuration, which is our *presumably optimal processor*. To evaluate the effectiveness of the input selection method, we compare the presumably optimal processor with the (globally) *optimal processor* obtained by taking into account all inputs during the design space exploration (left-hand side of Figure 4). We do this by computing normalized EDP for the presumably optimal processor, using Equation (4). The closer the normalized EDP is to 1, the closer the presumably optimal processor is to the (globally) optimal processor.

We consider four benchmark input selection methods, which we describe next.

### 5.1. Random Selection

To provide a solid ground of comparison, we first implemented *random selection*; that is, we select a number of inputs at random out of the pool of available benchmark inputs. More specifically, we evaluate the effect of the number of randomly selected inputs on

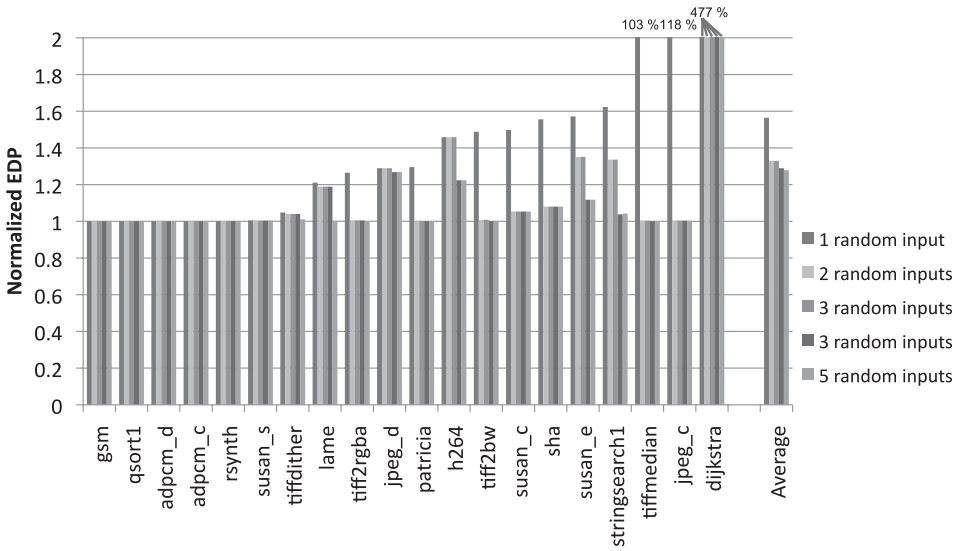


Fig. 5. Random input selection: normalized EDP as a function of number of randomly selected inputs.

the presumably optimal design. The expectation is that as we consider more randomly chosen inputs, the more representative this set of inputs becomes and hence the closer the presumably optimal processor will be to the globally optimal processor. This is indeed the case: As we increase the number of selected inputs from one to five, the average worst-case EDP decreases from 56% (one input), to 33% (two inputs), to 28% (five inputs), over the globally optimal processor; see Figure 5. These results were obtained by taking 1,000 randomly chosen sets of $n$ inputs, with $n$ varying from 1 to 5, and selecting the worst possible set. (Note that selecting one input at random corresponds to the results previously shown in Figure 2 under the "worst-case" scenario.)

As we increase the number of randomly selected inputs from one to two, we observe a fairly steep decrease in the average worst-case EDP from 56% to 33%, after which adding more inputs delivers diminishing returns. Some benchmarks greatly benefit from selecting a couple of randomly chosen inputs; see, for example, tiff, patricia, and jpeg_c. Unfortunately, there are also quite a few benchmarks for which selecting multiple inputs at random does not solve the problem; see, for example, lame, jpeg_d, dijkstra, and h264. Picking five randomly selected inputs may still lead to a suboptimal design point with an average 28% worst-case EDP deficiency compared to the globally optimal processor configuration. Clearly, random input selection is not effective at composing a representative input set, if few inputs are to be selected. Picking many more inputs is likely to solve this issue; however, it comes at the cost of requiring substantially longer simulation times during design space exploration. Instead, we devise other input selection methods that are more effective at selecting a few representative inputs than random selection.

### 5.2. Microarchitecture-Independent Selection

Our second input selection mechanism, *microarchitecture-independent selection*, selects a pair of representative inputs after characterizing the inputs in the input database through microarchitecture-independent characterization. Characterizing inputs incurs some overhead compared to random selection (as previously discussed, which incurs no overhead at all); however, the cost is a one-time cost only (i.e., each input needs

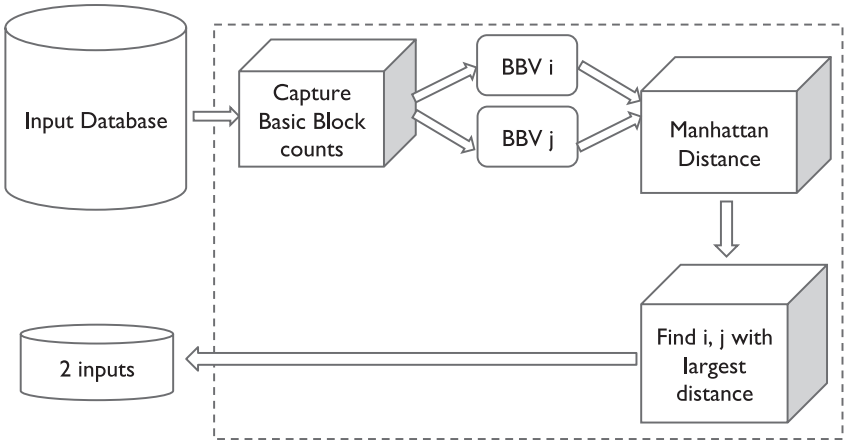


Fig. 6. Microarchitecture-independent input selection using Basic Block Vectors (BBVs).

to be characterized only once). Fortunately, this characterization step is much faster than detailed cycle-accurate simulation. The advantage of using a microarchitecture-independent characterization is that it characterizes the inputs in such a way that the characterization can be leveraged across microarchitectures and processor configurations. The microarchitecture-independent characterization used here is based on the concept of a Basic Block Vector (BBV), as previously introduced by Sherwood et al. [2001]. A basic block is an atomic piece of code with a single entry and exit point. A basic block vector is a vector that keeps track of how often each basic block is being executed dynamically. The dimensionality of a BBV equals the number of basic blocks in a program and each index in the BBV counts how many times the respective basic block is executed for a given input. In other words, a BBV represents which blocks of code are being executed, and how frequently. We normalize a BBV so that the absolute value of the vector equals one. We compute a BBV for each input and use it to characterize the dynamic behavior of the program and the given input in a microarchitecture-independent way. Previous work has demonstrated the good correlation between BBVs and dynamic execution behavior [Lau et al. 2005]. We build on this prior work and use BBVs to understand (dis)similarities among inputs.

The microarchitecture-independent input selection method is composed of three steps to create a reduced input set of two inputs; see also Figure 6. We first characterize each input through a BBV; that is, we execute the benchmark and its input and count how often each basic block is being executed—this can be done through functional simulation (using an architectural simulator such as `gem5`, as we did) or through a dynamic binary instrumentation tool such as Pin [Luk et al. 2005]. We subsequently normalize the BBVs. Next we calculate the Manhattan distance between the BBVs of each possible pair of inputs. The Manhattan distance between the BBVs of inputs $i$ and $j$ is calculated as follows:

$$MD(BBV_i, BBV_j) = \sum_{k=1}^{N} |BBV_i(k) - BBV_j(k)|, \tag{5}$$

with $N$ the dimensionality of the BBV, or the total number of static basic blocks in the program. In the third and last step, we determine the pair of inputs $(i, j)$ with the biggest Manhattan Distance (MD):

$$(i, j) = \arg\max_{i,j}(MD(BBV_i, BBV_j)). \tag{6}$$

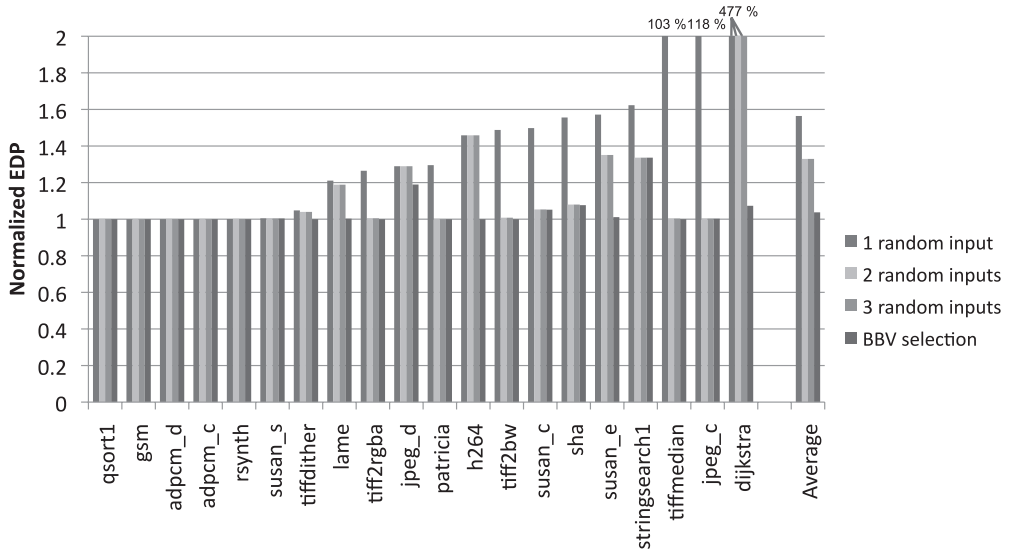


Fig. 7. Normalized EDP for microarchitecture-independent input selection using BBVs versus random selection.

Intuitively, this pair denotes inputs with the most divergent code execution behavior, which is likely to lead to diverse runtime behavior and is therefore likely to be more representative than randomly selected inputs.

Figure 7 compares the microarchitecture-independent selection using BBVs, as just described, against random selection. On average, this method results in a presumably optimal design point that is no more than 3.7% off compared to the globally optimal design point, which is a substantial improvement over random selection (e.g., 33% off, on average, for two randomly selected inputs). The maximum EDP deficiency for the BBV selection method is observed for stringsearch (33.6%); the reason we are seeing relatively high deficiencies using the BBV method (see jpeg_d and stringsearch) might be that a BBV only tracks the code being executed and not how it interacts with the microarchitecture (e.g., different memory access behavior may be observed even though the same code is being executed). For all applications, we are better off (or equally good) using BBV selection compared to randomly selecting two inputs. In fact, we even do better (or equally good) compared to three randomly selected inputs. This makes the microarchitecture-independent selection method using BBVs a reliable method for selecting representative inputs for design space exploration.

### 5.3. Filtered Selection

The third input selection method combines design space exploration and input filtering before randomly selecting a number of inputs. The basic idea is that if we can filter out nonrepresentative inputs, we can greatly reduce the worst-case scenario when randomly picking inputs. We have developed two filtered selection methods, which we describe next: one-level and two-level filtered selection. Both techniques use all inputs to perform a design space exploration in a limited subspace. Inputs that lead to far-from-optimal design points in the limited subspace are filtered out as it is likely that they are nonrepresentative of the larger design space. The resulting Filtered Input Database (FIDB) is then used to randomly select inputs.

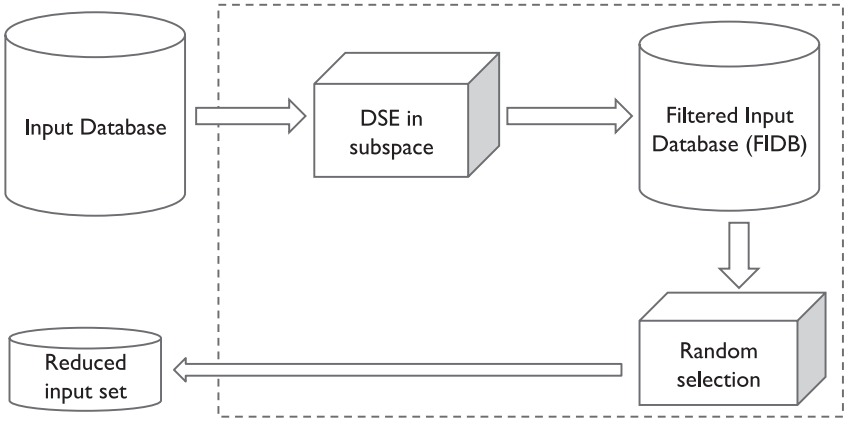


Fig. 8. One-level filtered input selection uses design space exploration in a limited subspace to filter out nonrepresentative inputs.

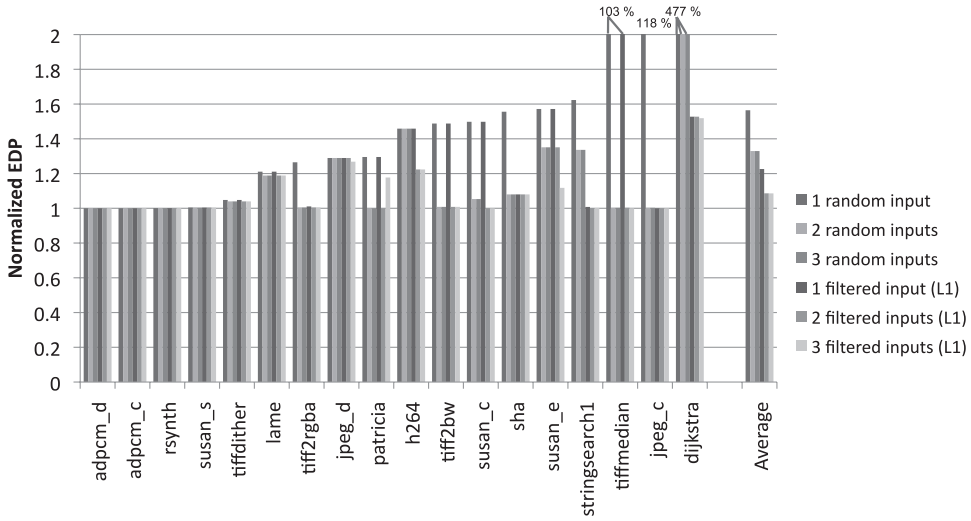


Fig. 9. Normalized EDP through one-level filtered input selection.

*5.3.1. One-Level Filtered Selection.* The one-level filtered selection method performs design space exploration with all the inputs on a single subspace of the larger design space, as shown in Figure 8. The design space exploration in this subspace involves estimating performance and power/energy for all design points in the subspace and for all inputs in the input database. We consider a subspace of 10 design points; these design points include a baseline configuration (conf0 from Table I) along with nine other design points derived from this baseline by randomly changing a number of the microarchitecture parameters shown in Table II, one at a time. Once the exhaustive evaluation is done in this subspace, we filter out the inputs that result in poor average normalized EDP, relative to the optimum design point in the subspace. This results in an FIDB. The hope then is that the FIDB no longer contains inputs that may lead to poor design points in the larger design space. From the FIDB, we then select a number of inputs at random.

Figure 9 reports worst-case normalized EDP for the one-level filtered selection method when randomly picking one, two, and three inputs from the FIDB. One-level

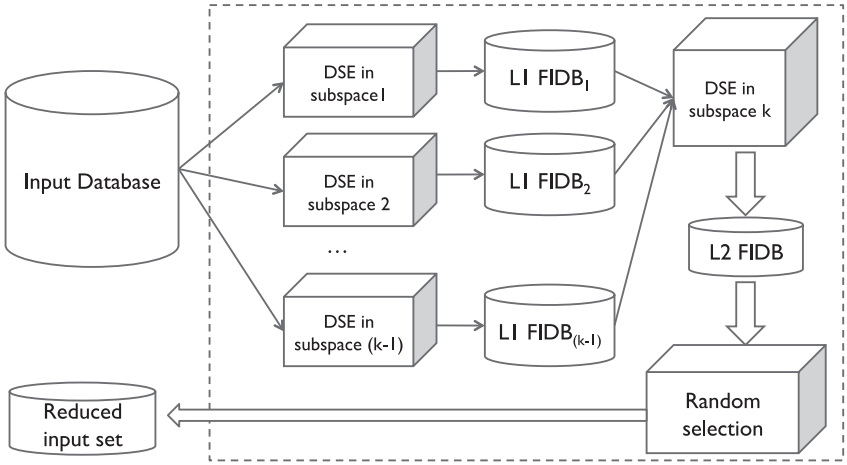


Fig. 10. Two-level filtered input selection uses two levels of design space exploration in limited subspaces to filter out nonrepresentative inputs.

filtered input selection leads to average worst-case EDP deficiencies of 22.6%, 8.6%, and 8.1%, respectively, which is a substantial improvement over random selection. This confirms that performing a preliminary design space exploration in a small subspace of the larger design space filters out a number of nonrepresentative inputs. For most benchmarks, between 0 and 200 inputs are filtered out; that is, the FIDB contains between 800 and 1,000 inputs. However, for two benchmarks (dijkstra and jpeg_c), the FIDB contains slightly more than half of the original input database. The cost of one-level filtered selection is a one-time cost involving detailed simulation of all design points in the subspace for all inputs. In our setup with 1,000 inputs and 10 design points in the subspace, this results in 10,000 evaluations per benchmark.

*5.3.2. Two-Level Filtered Selection.* To further improve the effectiveness of the filtered input selection method, we now consider two levels of filtering; see also Figure 10. As with the one-level filtered selection method, we first rely on a design space exploration in a limited subspace. However, instead of selecting a single subspace, we select $k$ subspaces. We randomly divide the original input database in $(k-1)$ sets of inputs and we perform an exhaustive evaluation in $(k-1)$ subspaces. This leads to $(k-1)$ level-one (L1) FIDBs. Next, we randomly select a number of inputs from each of the FIBDs, which we use to exhaustively evaluate the $k$th subspace. This leads to a second level of filtering, yielding the level-two (L2) FIDB. We then select a number of inputs at random from the L2 FIDB.

In our implementation, we aimed at having the same number of simulations as for one-level filtered input selection, namely, 10,000 evaluations per benchmark. We therefore set $k$ to 10 and simulate 100 inputs per subspace. Each subspace is randomly selected and consists of 10 design points. For most benchmarks the L1 FIDBs contain at least 56 inputs of the original 100 inputs. One benchmark, sha, has an L1 FIDB of 17 inputs only. Some benchmarks have a couple of L1 FIDBs with around 60 inputs only, while the other FIDBs contain around 100 inputs. This implies that the selected subspace influences the selection of inputs and having multiple subspaces could potentially lead to better results than using a single subspace. To construct the L2 FIDB, we select 100 random inputs across all the L1 FIDBs. The size of the L2 FIDB ranges between 56 and its maximum possible value of 100.

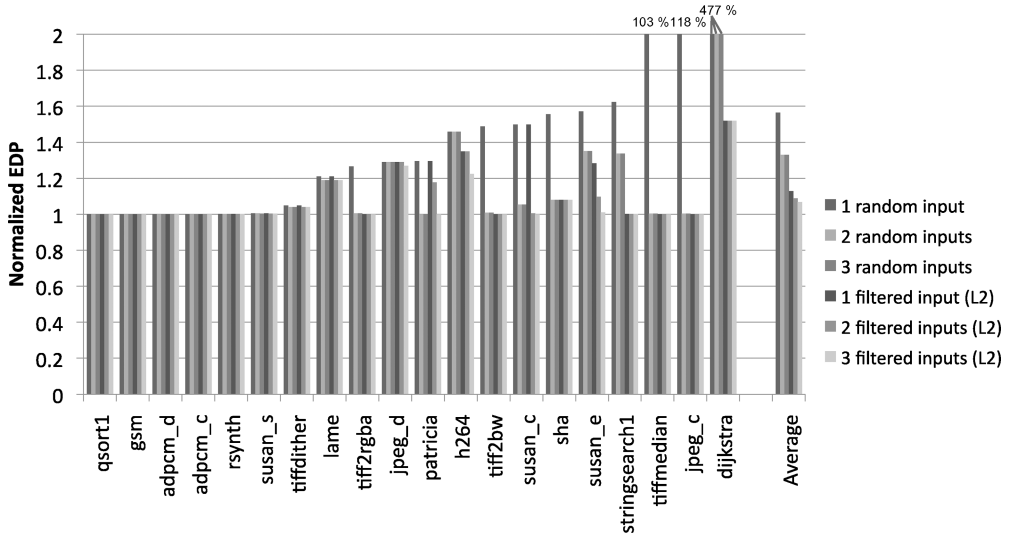


Fig. 11. Normalized EDP through two-level filtered input selection.

Figure 11 reports that picking one random input from the L2 FIDB leads to an average worst-case EDP deficiency of 12.9% over the globally optimal design point, which is a substantial improvement over random selection and one-level filtered selection. This implies that two-level filtered selection effectively does a better job filtering out nonrepresentative inputs. Randomly picking two and three inputs from the L2 FIBD leads to an average worst-case EDP deficiency of 8.7% and 6.7%, respectively.

It is worth noting that the input filtering approach, while effective on average, is not perfect. We observe a clear benefit from filtering for all benchmarks, except for patricia. In particular, for patricia, when considering two inputs, random selection outperforms two-level filtering; similarly, for three random inputs, random selection outperforms one-level filtering; likewise, for two inputs, one-level outperforms two-level filtering. This counterintuitive result suggests that we may be discarding representative inputs, which may be due to the small subspaces used to filter out inputs; that is, the subspaces may not be representative for the larger space. A systematic way of determining subspaces may solve this issue; however, we leave this for future work as the random subspace selection process seems to work well for most benchmarks in our study.

### 5.4. Min-Median-Max Selection

The filtered input selection method, while effective, has two major shortcomings. First, the cost for building the FIDB is quite high; that is, it requires detailed evaluation of all inputs on a number of design points. Second, there is a random component to the selection of inputs from the FIDB. We now propose an input selection method, called *min-median-max selection*, that overcomes these issues. It involves a single evaluation of a particular design point only and selects representative inputs in a systematic way, not through random selection.

The min-median-max selection method was inspired by filtered selection, but instead of evaluating subspaces, we consider a single baseline design point only, on which we evaluate all inputs. As we have a single design point only, we have no means of filtering out nonrepresentative inputs based on their performance with respect to ranking design points in the subspaces. We therefore resort to relative performance of the input compared to the other inputs in the input database; that is, we pick inputs

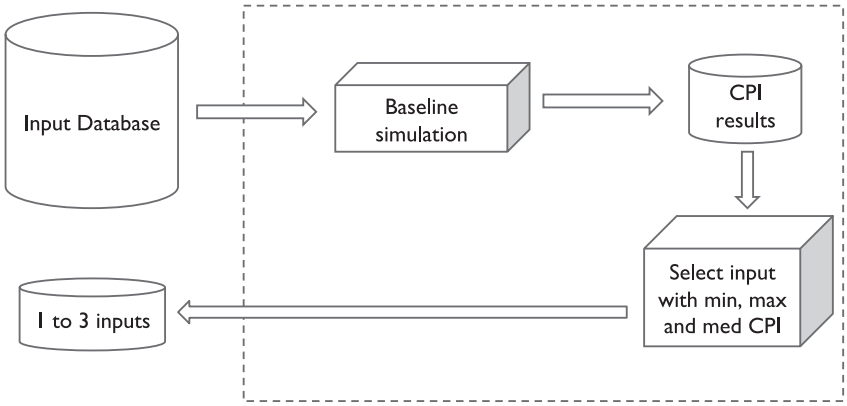


Fig. 12. Min-median-max selection.

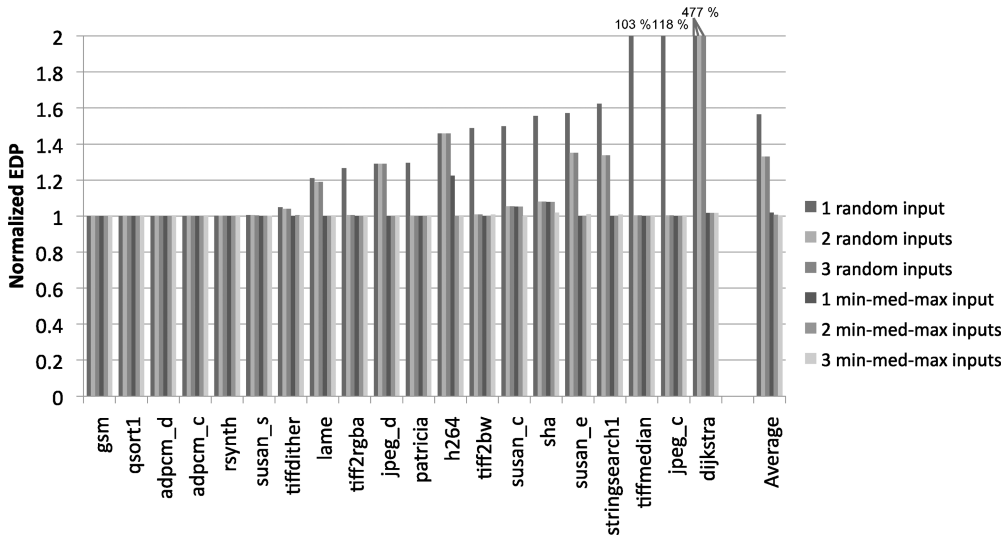


Fig. 13. Normalized EDP through min-median-max selection.

based on the achieved Cycles Per Instruction (CPI) on the baseline architecture. When selecting one input, we pick the input that has the median CPI across all inputs; when selecting two representative inputs, we pick the input with median CPI and another input that has a value as close as possible to the median CPI;[1] when selecting three inputs, we pick the inputs with the minimum, median, and maximum CPI value. See Figure 12 for a schematic overview of the min-median-max selection method.

The min-median-max method is effective at identifying representative inputs, as shown in Figure 13 for one input (with median CPI), two inputs (with median CPI and nearly median CPI), and three inputs (minimum, median, and maximum CPI). One input leads to an average worst-case EDP deficiency of 2% compared to the globally optimal design point (with one outlier of 22% for h264). Two inputs brings the worst-case EDP deficiency further down to 1%, on average (and at most 7% for sha). Three

[1]We found selecting two inputs close to the median CPI to outperform selecting the inputs with the minimum and maximum CPI.

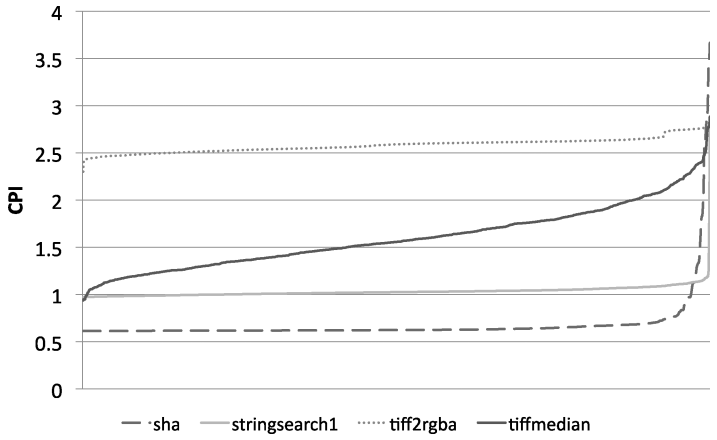


Fig. 14. CPI across datasets for a number of benchmarks.

Table IV. Summary of Input Selection Methods in Terms of Their One-Time Cost, Number of Inputs Selected, Average Worst-Case EDP Deficiency, and Maximum Worst-Case EDP Deficiency, Where *N* is the Number of Inputs in the Database

| *Selection method* | *One-time cost* | *# Inputs* | *Avg worst case* | *Max worst case* |
|---|---|---|---|---|
| Random selection | None | 1 - 2 - 3 | 56% - 33% - 33% | 477% - 477% - 477% |
| One-level filtering | $10 \times N$ detailed evaluations | 1 - 2 - 3 | 23% - 9% - 8% | 103% - 53% - 52% |
| Two-level filtering | $10 \times N$ detailed evaluations | 1 - 2 - 3 | 13% - 9% - 7% | 52% - 52% - 52% |
| BBV selection | Instrumentation of $N$ inputs | 2 | 4% | 34% |
| Min-median-max | $N$ detailed evaluations | 1 - 2 - 3 | 2% - 1% - 0% | 22% - 8% - 2% |

inputs leads to no EDP deficiency; that is, the three representative inputs lead to the same optimum design point as using all inputs. The intuition is that the three most extreme inputs, the ones with the minimum, median, and maximum performance on a baseline design point, are representative of the entire input database—the other inputs can be considered "interpolations" with respect to the extreme inputs—which leads to effective design space exploration.

Figure 14 shows CPI values for the baseline configuration of a number of benchmarks across all of their respective inputs and provides intuition why for some of the benchmarks it is better to pick inputs with median and near-median CPI rather than inputs with minimum and maximum CPI. Benchmarks such as tiffmedian and tiff2rgba (and some others that are not displayed due to space reasons) that have an S-shaped curve for their CPI values tend to have good results when selecting inputs with minimum and maximum CPI. Benchmarks sha and stringsearch (and others not displayed due to space reasons), on the other hand, are L-shaped, and selecting inputs with minimum and maximum CPI tend to perform poorly on these benchmarks. The reason is that there are very few inputs with a relatively high CPI. Selecting these inputs puts too much weight on these inputs that occur infrequently and are nonrepresentative for the rest of the inputs. Adding the inputs with median CPI addresses this issue.

### 5.5. Overview and Discussion

Table IV summarizes and compares the proposed input selection methods. While random selection incurs no cost for selecting inputs, it may lead to severe EDP deficiencies. Even selecting three inputs at random may result in an average worst-case EDP deficiency of 33% compared to using all inputs. The filtering method is more effective at selecting representative inputs, with an average worst-case EDP deficiency of 6.7% for

the two-level approach and three randomly selected inputs. However, filtering comes at the cost of requiring $10 \times N$ detailed evaluations (detailed simulations), with $N$ the number of inputs in the input database. The microarchitecture-independent selection method using BBVs incurs a one-time cost of collecting BBVs for all inputs. Fortunately, collecting BBVs is essentially a profiling step, which is much faster than detailed simulation. The BBV selection method is more effective than filtering. It achieves an average worst-case EDP deficiency of 3.7% while using only two inputs during design space exploration; we observed an EDP deficiency of 33.6% for one benchmark though. Min-median-max selection is the most effective approach: Selecting the inputs with the minimum, median, and maximum CPI for a baseline configuration leads to identifying the globally optimal design point for all benchmarks. This comes at the cost of requiring $N$ detailed evaluations, which is more time-consuming than BBV profiling.

These results lead to two important conclusions or insights. First, it is important to select benchmark inputs in a systematic way. Randomly selected inputs, even from a subset of inputs as with the filtering approach, may lead to nonrepresentative behavior, which eventually leads to nonoptimal design points during design space exploration. Selecting inputs based on their behavior, either through microarchitecture-independent characterization (e.g., BBV selection) or through detailed measurements on a particular system (e.g., min-median-max), improves the representativeness of the workload, ultimately leading to a better final design. Second, the two most accurate input selection methods involve a tradeoff. Min-median-max selection is (slightly) more effective compared to BBV selection at identifying the optimum design point; however, this comes at the cost of incurring more overhead as it requires detailed measurements for all inputs on a given baseline design point.

Although the presented techniques have only been evaluated for superscalar in-order processors, we believe that they also apply (and might be even more important) for more complex microarchitectures (e.g., out-of-order processors). Performance might be more sensitive to inherent workload behavior and input selection as the architecture is more complex; hence, a broader set of inputs might be needed compared to in-order processors. Some of the proposed techniques might need minor adjustments to select a slightly broader range of representative inputs. For example, the min-med-max selection method can be extended by adding inputs between the minimum and median, and between the median and maximum, to have a more fine-grained selection of inputs. When looking at multicore processors, special attention should be given regarding how to best select inputs to form representative multiprogram and multithreaded workloads. Prior work has selected multiprogram workloads randomly [Van Craeynest and Eeckhout 2011] or based on microarchitecture-independent characterization of benchmarks [Van Biesbrouck et al. 2007]. Extending and/or combining these techniques with techniques for selecting representative inputs is left for future work.

## 6. RELATED WORK

It is well known that having representative benchmarks is key to design space explorations, as using nonrepresentative benchmarks may lead, or is likely to lead, to suboptimal design points. As a result, substantial prior work has been done toward identifying representative benchmarks; however, to the best of our knowledge, no comprehensive study was previously published regarding the impact of benchmark inputs on design space exploration.

We now discuss some of the prior work in benchmark selection, followed by a description of prior work in generating and evaluating benchmark inputs and design space exploration methods.

### 6.1. Benchmark Selection

Common practice when composing a benchmark suite is to identify key applications in a given application domain of interest, from which benchmarks and inputs are then selected. An important requirement for benchmark suites to be shared across parties (academia and/or industry) is that the benchmarks are open source, although NonDisclosure Agreements (NDAs) may enable processor manufacturers to use proprietary customer workloads to evaluate future designs. Example open-source benchmark suites are SPEC CPU [Henning 2006] for general-purpose computing, DaCapo [Blackburn et al. 2006] for Java workloads, PARSEC [Bienia et al. 2008] for multicore, Rodinia [Che et al. 2009] for heterogeneous CPU/GPU systems, MiBench [Guthaus et al. 2001] for embedded workloads, CloudSuite [Ferdman et al. 2012] for cloud workloads, and so forth. The benchmark selection typically involves making sure the benchmark suite, as a whole, covers the most important application behaviors in the target domain while being portable enough to use across different platforms.

A number of papers have been published to evaluate the representativeness of a benchmark suite and/or to select a representative subset from a larger pool of benchmarks. Eeckhout et al. [2002] propose data analysis (principal components analysis) and machine learning (cluster analysis) techniques to identify a diverse and representative benchmark subset. Follow-on work used microarchitecture-independent benchmark characterization as input to this methodology [Joshi et al. 2006] or a number of real hardware measurements [Phansalkar et al. 2007]. Yi et al. [2003] use a Plackett-Burman design of experiment in which they measure performance on a number of different processor architectures to understand how differently benchmarks interact with microarchitecture parameters. They then pick diverse benchmarks using cluster analysis to cover the workload space as much as possible. SubsetTrio [Jin and Cheng 2011] translates the benchmark selection problem into a geometrical problem and uses the notion of a convex hull to identify most diverse benchmarks—the benchmarks at the outer range of the benchmark space—in contrast to cluster analysis, which groups benchmarks into clusters of similar benchmarks and then picks a representative benchmark per cluster. Yi et al. [2006] quantify the pitfall of using nonrepresentative, old benchmarks to design future processors. They discuss a case study in which they identify the optimum processor for SPEC CPU95 and then evaluate this processor using CPU2000. They report an EDP deficiency of 18.5%. This, once more, underlies the importance of using representative benchmarks during design space exploration.

While composing representative benchmark suites is challenging for single-core experiments, it is a daunting task for multicore and multithreaded processors. Such processors can run multiple independent thread contexts, which leads to an explosion in the number of possible multiprogram workloads [Van Biesbrouck et al. 2007; Van Craeynest and Eeckhout 2011; Velasquez et al. 2013] and combinations of benchmark starting points [Van Biesbrouck et al. 2006; Sandberg et al. 2013].

In addition to finding representative benchmarks, an additional concern is to find regions within a benchmark's execution that are representative for the entire benchmark execution. SimPoint [Sherwood et al. 2002] collects BBVs on a per-region basis and uses cluster analysis to find representative regions within a benchmark execution. Eeckhout et al. [2005] use microarchitecture-independent characterization to identify representative regions across benchmarks. Van Biesbrouck et al. [2004] introduce the Co-Phase Matrix to find the most representative regions in multiprogram workloads.

### 6.2. Input Selection

As mentioned before, the amount of work done in characterizing the impact of input datasets on design space exploration is limited. KleinOsowski and Lilja [2002]

proposed reduced input sets for SPEC CPU2000 to reduce simulation time. Later work by Eeckhout et al. [2003] found these reduced inputs to be representative for some benchmarks, but not for others. Chen et al. [2010] propose KDataSets, a set of 1,000 datasets for a broad set of the MiBench benchmarks. They use KDataSets to understand dataset sensitivity for iterative optimization, which aims at finding the best set of compiler optimizations for a given application. Breughe et al. [2011] uses KDataSets to study how sensitive processor customization is with respect to application inputs. While this paper correctly states that using a single (or just a few) input(s) is sufficient to determine a processor design optimal for a specific application—see the "best case" results in Figure 2—it does not provide a method to select this input.

### 6.3. Design Space Exploration

Design space exploration is a very complicated endeavor for a number of reasons. For one, the design space is typically huge. Second, the evaluation of a single design point typically takes a very long time due to the slow simulation speeds of detailed cycle-accurate simulators. Given the importance of the problem, a fair amount of research has been done on design space exploration. In particular, Yi et al. [2003] use a Plackett-Burman design of experiment to identify the important axes in the design space in order to drive the design process. Eyerman et al. [2006] leverage machine learning techniques and optimization strategies such as genetic algorithms to more quickly steer the search process to the optimum design point. Lee and Brooks [2008] use empirical models to explore the design space of adaptive microarchitectures, while Karkhanis and Smith [2007] use a mechanistic model to guide the design of application-specific out-of-order processors.

## 7. CONCLUSION

In this article, we provide a comprehensive study to emphasize the sensitivity of design space exploration to benchmark inputs. Because using a large number of inputs to drive design space exploration is infeasible, we propose a number of methods to find a reduced set of inputs. The representativeness of the set of inputs produced by these methods is evaluated in the article.

Our setup to study input sensitivity for design space exploration involves 1,000 inputs for 20 embedded benchmarks and a design space consisting of around 1,700 design points. We use EDP as our optimization criterion, and we focus on the average EDP deficiency across all inputs, when the worst possible input(s) are used during design space exploration. This is a case that should be avoided at all cost because it is unknown a priori whether the selected inputs are representative or not. Using a single randomly selected input, the design space exploration could lead to an average worst-case EDP deficiency of 56% compared to the design point with minimum EDP. When using three randomly selected inputs, the worst-case EDP deficiency can be reduced to 33%. However, these high-deficiency numbers can be greatly reduced by the three methods proposed in this article: filtered input selection, input selection through microarchitectural-independent characterization (BBV selection), and min-median-max selection. If we apply two levels of filtering before randomly selecting three inputs, we reduce the worst-case EDP deficiency to 6.7% on average. BBV selection achieves an EDP deficiency of 3.7% with two inputs selected; min-median-max selection finds the design point with minimum EDP (i.e., 0% deficiency) by using no more thanr three inputs. BBV selection and min-median-max selection are more effective than filtered selection while incurring a lower overall cost.

Overall, we hope this article increases awareness to benchmark inputs and its importance to design space exploration. As nowadays architecture research and development is increasingly data driven, it requires researchers and designers to pay close

attention to selecting representative benchmark inputs. We believe this is going to be even more necessary as we resort to application-specific and domain-specific specialization of processor hardware to sustain the performance and energy-efficiency growth curve in the absence of Dennard scaling.

## REFERENCES


Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The PARSEC Benchmark Suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. 2011. The gem5 simulator. *Computer Architecture News 39*, 2, 1–7.

Blackburn, S. M., Garner, R., Hoffmann, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A. L., Jump, M., Lee, H. B., Moss, J. E. B., Phansalkar, A., VanDrunen, T., von Dincklage, D., and Wiedermann, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*. 169–190.

Breughe, M., Eyerman, S., and Eeckhout, L. 2012. A mechanistic performance model for superscalar in-order processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'12)*. 14–24.

Breughe, M., Li, Z., Chen, Y., Eyerman, S., Temam, O., Wu, C., and Eeckhout, L. 2011. How sensitive is processor customization to the workload's input datasets? In *Proceedings of the IEEE International Symposium on Application-Specific Processors (SASP'11)*. 1–7.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. 44–54.

Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., and Wu, C. 2010. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 448–459.

Dennard, R. H., Gaensslen, F. H., Yu, H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid-State Circuits 9*, 5, 256–268.

Eeckhout, L., Sampson, J., and Calder, B. 2005. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization (IISWC'05)*. 2–12.

Eeckhout, L., Vandierendonck, H., and De Bosschere, K. 2002. Workload design: Selecting representative program-input pairs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'02)*. 83–94.

Eeckhout, L., Vandierendonck, H., and De Bosschere, K. 2003. Designing workloads for computer architecture research. *IEEE Comput. 36*, 2, 65–71.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 365–376.

Eyerman, S., Eeckhout, L., and De Bosschere, K. 2006. Efficient design space exploration of high performance embedded out-of-order processors. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE'06)*. 351–356.

Ferdman, M., Adileh, A., Kocberber, Y. O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 37–48.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC'01)*. 3–14.

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B. C., Richardson, S., Kozyrakis, C., and Horowitz, M. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. 37–47.

HENNING, J. L. 2006. SPEC CPU benchmark descriptions. *SIGARCH Comput. Archit. News 34*, 4, 1–17.

JIN, Z. AND CHENG, A. C. 2011. SubsetTrio: An evolutionary, geometric and statistical benchmark subsetting approach. *ACM Trans. Model. Comput. Simul. 21,* 3.

JOSHI, A., PHANSALKAR, A., EECKHOUT, L., AND JOHN, L. K. 2006. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput. 55*, 6, 769–782.

KARKHANIS, T. AND SMITH, J. E. 2007. Automated design of application specific superscalar processors: An analytical approach. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. 402–411.

KLEINOSOWSKI, A. J. AND LILJA, D. J. 2002. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Comput. Archit. Lett. 1*, 2, 10–13.

LAU, J., SAMPSON, J., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2005. The strong correlation between code signatures and performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*. 236–247.

LEE, B. AND BROOKS, D. 2008. Efficiency trends and limits from comprehensive microarchitectural adaptivity. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. 36–47.

LI, S., AHN, J. H., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 469–480.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'05)*. 190–200.

PHANSALKAR, A., JOSHI, A., AND JOHN, L. K. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 Benchmark Suite. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA'07)*. 412–423.

SANDBERG, A., SEMBRANT, A., HAGERSTEN, E., AND BLACK-SCHAFFER, D. 2013. Modeling performance variation due to cache sharing. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'13)*. 155–166.

SAZEIDES, Y., KUMAR, R., TULLSEN, D. M., AND CONSTANTINOU, T. 2005. The danger of interval-based power efficiency metrics: When worst is best. *IEEE Comput. Archit. Lett. 4,* 1.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*. 3–14.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. 45–57.

VAN BIESBROUCK, M., EECKHOUT, L., AND CALDER, B. 2006. Considering all starting points for simultaneous multithreading simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'06)*. 143–153.

VAN BIESBROUCK, M., EECKHOUT, L., AND CALDER, B. 2007. Representative multiprogram workloads for multithreaded processor simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'07)*. 193–203.

VAN BIESBROUCK, M., SHERWOOD, T., AND CALDER, B. 2004. A co-phase matrix to guide simultaneous multithreading simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*. 45–56.

VAN CRAEYNEST, K. AND EECKHOUT, L. 2011. The multi-program performance model: Debunking current practice in multi-core simulation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'11)*. 26–37.

VELASQUEZ, R. A., MICHAUD, P., AND SEZNEC, A. 2013. Selecting benchmark combinations for the evaluation of multicore throughput. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*.

VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., AND TAYLOR, M. B. 2010. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. 205–218.

Xiph.org. 2012. Video Test Media—derf's collection. http://media.xiph.org/video/derf/.

YI, J. J., LILJA, D. J., AND HAWKINS, D. M. 2003. A statistically rigorous approach for improving simulation methodology. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA'03)*. 281–291.

YI, J. J., VANDIERENDONCK, H., EECKHOUT, L., AND LILJA, D. J. 2006. The exigency of benchmark and compiler drift: Designing tomorrow's processors with yesterday's tools. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS'06)*. 75–86.

Received June 2013; revised September 2013; accepted September 2013